MSc thesis

Computer Science

Opportunities and challenges in adopting continuous end-to-end testing: A case study

Atte Lassila

December 7, 2019

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. Tommi Mikkonen

Examiner(s)

Dr. Niko Mäkitalo

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Computer Science

Tekijä — Författare — Author

Atte Lassila

Työn nimi — Arbetets titel — Title

Opportunities and challenges in adopting continuous end-to-end testing: A case study

Ohjaajat — Handledare — Supervisors

Prof. Tommi Mikkonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	December 7, 2019	44 pages

Tiivistelmä — Referat — Abstract

Modern software systems increasingly consist of independent services that communicate with each other through their public interfaces. Requirements for systems are thus implemented through communication and collaboration between different its services. This creates challenges in how each requirement is to be tested.

One approach to testing the communication procedures between different services is end-to-end testing. With end-to-end testing, a system consisting of multiple services can be tested as a whole. However, end-to-end testing confers many disadvantages, in tests being difficult to write and maintain. When end-to-end testing should adopted is thus not clear.

In this research, an artifact for continuous end-to-end testing was designed and evaluated it in use at a case company. Using the results gathered from building and maintaining the design, we evaluated what requirements, advantages and challenges are involved in adopting end-to-end testing.

Based on the results, we conclude that end-to-end testing can confer significant improvements over manual testing processes. However, because of the equally significant disadvantages in end-to-end testing, their scope should be limited, and alternatives should be considered. To alleviate the challenges in end-to-end testing, investment in improving interfaces, as well as deployment tools is recommended.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Software verification and validation

Avainsanat — Nyckelord — Keywords

continuous integration, testing

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software Systems study track

Contents

1 Introduction

Modern software development practices such as DevOps emphasize that software projects should be continuously and automatically built, tested, and deployed (Smeds et al., 2015). At the same time, there is an increasing trend away from monolithic software architectures. When those features are implemented via collaboration of multiple services, automatic testing can become more complex, as testing a requirement fully might require execution on multiple independent services.

Testing communication and collaboration between multiple independent services is called end-to-end testing. End-to-end testing tests functionality at the granularity of the whole system. Testing at this granularity can require extensive setup, and the tests are often much more burdensome to write and maintain than lower level tests such as unit tests. (Wacker, 2015)

As more features are implemented through collaboration of multiple services, there consequently needs to more of a focus on implementing and maintaining tests for that communication. End-to-end tests are one way of accomplishing this. (Clemson, 2018)

In this research, we present a design for an automated end-to-end testing artifact for a data integration system used at a Finnish software company. The system serves as an integration point for customers. Customers data is received and transformed by the system to a form readable by the company's core product. Because of the system's importance in the customers' daily operations, it is important to verify that there are no regressions in the communication with the system and the core product. The designed artifact is evaluated to find out what requirements, advantages, and challenged there are involved in adopting continuous end-to-end testing.

Rest of the paper is organized as follows. In Chapter 2, we dive into the literature concerning end-to-end testing and continuous practices. Chapter 3 concerns the high-level goals of the research, whereas Chapter 4 describes the case project and the requirements for the testing artifact for it. From there, Chapter 5 describes the design of the artifact. Chapter 6 considers the results of the design and whether it fulfills the requirements. Chapter 7 discusses the results in depth, stating how they answer the research goals. Alternative approaches and future work is also posited in this chapter. Finally, in Chapter 8, we summarize the research, and offer brief conclusions.

2 Background

Modern software development practices increasingly emphasize the importance of providing developers with quick feedback for changes (Fitzgerald and Stol, 2014). Automatic testing and continuous development practices are important tools for achieving this. In this chapter, we first explore software testing from this perspective. From there, we detail the most important continuous software development practices, and how they enable quicker feedback cycles. Finally, we explain data integration, giving some context to the case project evaluated in this thesis.

2.1 Testing

Software testing is how software projects are verified to work correctly. Tests serve as requirements imposed on a project. With automatic testing, these requirements are expressed as code, which can be rerun against the system in the event of any and all changes to its components. Tests are often divided into categories of higher and lower *test granularities*, with *end-to-end testing* being an example of a higher-granularity test.

2.1.1 Test granularity

Test granularity refers to the level of detail in software tests. On lower levels of test granularity, the scope of the tests is very small. There, tests can test very detailed requirements. Unit tests are the primary example of testing at a lower granularity. As tests reach higher levels of granularity, the number of involved components increases, and the level of detail in the test cases is lowered. Tests on the higher granularity levels test at the level of an entire system. (Rothermel et al., 2002)

In comparison to lower granularity tests, higher granularity tests are often much more time-consuming to write and to maintain. Setting up a suitable test environment takes up a larger portion of the test runtime. As higher granularity tests include more components, the tests themselves can also take more time to execute. The increased number of components can lead to there being more asynchronous communication and

lengthy calculations that the tests will have to wait for. (Rothermel et al., 2002)

Because of the difference in cost-effectiveness, is is recommended that lower granularity tests make up the majority of a system's tests, as suggested in Figure 2.1 (Clemson, 2018). Manual exploratory testing should make up a small minority of a mature system's testing strategy. End-to-end testing can be useful but should be limited. Component tests for individual services are great for testing a system in isolation, but can miss lower level details. Integration and unit tests meanwhile, should be the core of any system's testing strategy, as they are much better at isolating failures.

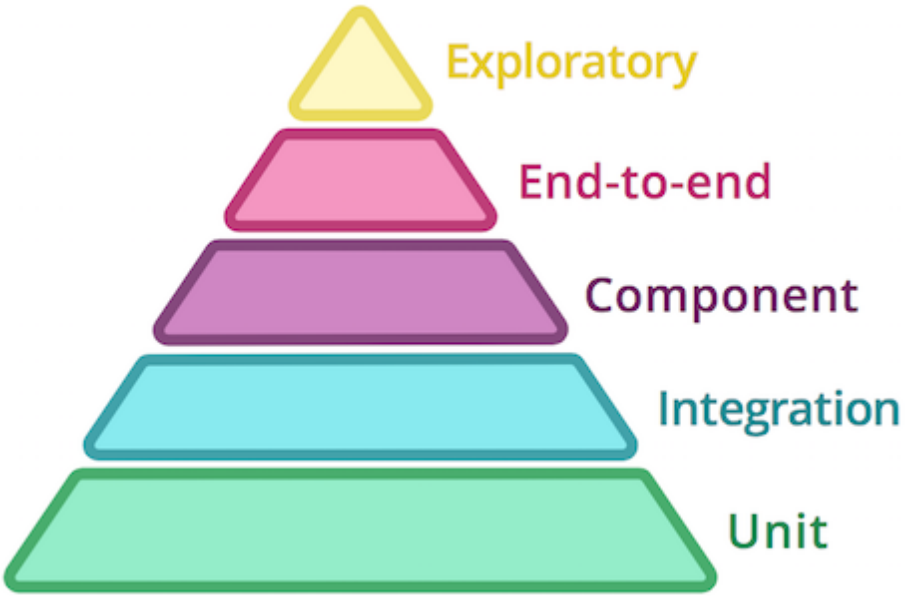

Figure 2.1: Lower granularity tests should make up the majority of a system's tests, with higher granularity tests kept to a minimum (Clemson, 2018).

2.1.2 End-to-end testing

End-to-end testing is testing a set of services for whether they communicate correctly with each other. With higher-granularity end-to-end tests, a system or a subset of the services that make up the system can be tested as a whole. (Clemson, 2018)

Depending on the practitioner and software domain, end-to-end testing can take slightly different meanings. In the context of this research, end-to-end testing is a tool to verify that the communication procedures between independent services work as intended. Here, we will give a few examples of end-to-end tests, as well as discuss the challenges inherent to end-to-end testing.

Examples. End-to-end testing is particularly relevant for projects where features are implemented via communication between multiple components, such as projects using microservice architecture. Generally each microservice communicates with other services via well-defined web interfaces. With end-to-end testing, it can be verified that the procedures for each service communicating through the other services' interfaces are *correct*. (Clemson, 2018)

A common example of end-to-end testing is testing that the web user interface and server backend of an application work in tandem correctly from the user's perspective (Fowler, 2013). In this type of testing, rather than testing communication procedures written down in code, the tests simulate real user workflows. End-to-end testing through user interfaces has the most tools to support it, Cypress (Cypress, 2019) and Selenium (Selenium, 2019) being common tools used.

For an another example, consider a microservice architecture implementation for an online store. One microservice might be in charge of keeping track of the inventory, another of the users, and one would handle payment processing. In this case, finding out which users have unpaid purchases for which products becomes a task encompassing multiple services. Lower level tests could make sure that each service works correctly in isolation, but could not guarantee that they communicate in the correct fashion to find the result. One way to add that guarantee is end-to-end testing.

Challenges. End-to-end tests are often the highest granularity automated tests a system has. For this reason, the problems that higher granularity tests have are especially noticeable in end-to-end testing. This is shown in the tests breaking more often and having a long runtime.

As more and more systems and components are included in a test, the tests become more susceptible to breaking when changes are made to any of the components included. This is called a test being *flaky*. By increasing the scope of tests, tests can better test the integration of multiple components. In doing so however, the tests become more flaky. Conversely, when the scope of a test is limited, they generally break less often. The scope being limited also means that test failures are much easier to isolate to the failing component. (Clemson, 2018)

When tests are on the level of a whole system and its independent services, setting up a test environment will also require more effort. If a system's individual components or their deployment processes are complex, this can result in flaky test environment. A flaky test environment can cause test failures that originate not from the test cases

failing, but the test environment not producing reproducible results (Wacker, 2015).

As an end-to-end test often targets a full system, with all or a subset of its services, the tests are required to orchestrate those systems such that each service is running, and in a clean slate before each test case. Communication between tested services can additionally be asynchronous or include long-running processes. These contribute to the slower runtime of end-to-end tests as compared to lower granularity testing.

Because of the limitations for end-to-end testing, it is recommended that the number of end-to-end tests compared to the number of lower granularity tests is limited to *as few as possible*. In some cases, it might be possible to omit end-to-end testing completely, if confidence in the communication between services is achieved in a different way. (Fowler, 2013)

2.2 Continuous practices

Continuous practices generally refers to the practices of continuous integration (*CI*), delivery (*CDe*) and deployment (*CD*), which make up a part of the whole of continuous engineering (Fitzgerald and Stol, 2014). In general, continuous practices encourage performing certain parts of project development and release often and consistently during a project's life span. In this chapter we explain each of the practices of CI, CD, and CDe, as well as which tools are used for implementing them. Figure 2.2 shows which software development tasks these continuous practices influence.

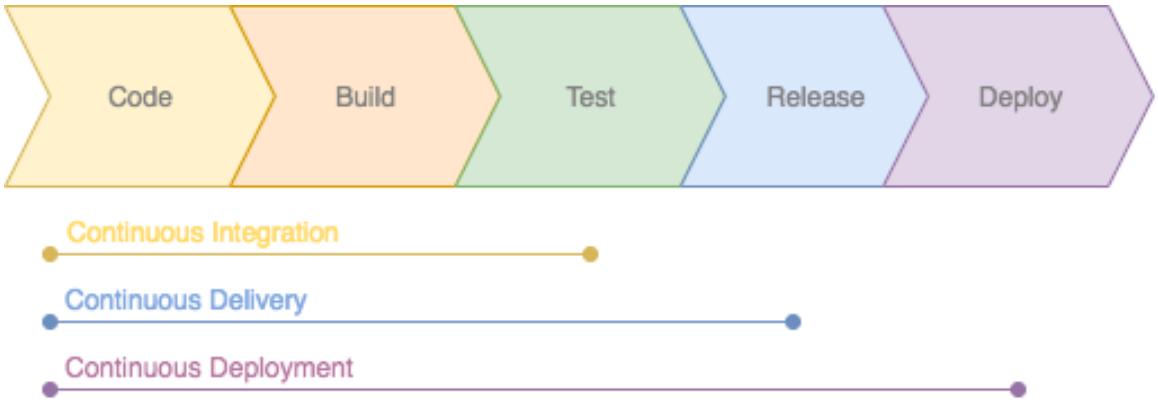

Figure 2.2: Continuous practices and which software development tasks they encompass.

2.2.1 CI, CD & CDe

Continuous integration, or **CI** is the process of continuously merging and testing changes by different developers and teams (Fowler and Foemmel, 2006). In the past, integration used to be a lengthy process involving the exchange of floppy disks, manual conflict management and a long testing period. Commonly, integration was done very infrequently, which further exacerbated the problems.

By practicing CI, software teams commit to integrating changes as often as multiple times per day. This is made possible with an assortment of developer tools which automate source code management, builds and testing. CI's main characteristic is short *feedback cycles* to developers, informing them of when the changes they have made conflict with or break existing functionality.

Continuous deployment, or **CD**, aims to automate the deployment of changes to production. In continuous deployment, changes are automatically and without much manual action deployed to production environments. This allows developers to gain much quicker user feedback for changes they have developed, potentially improving quality and development speed. (Humble and Farley, 2010)

Continuous delivery, or **CDe** is the process of automating the methods for bringing a software project onto a production environment (Humble, 2010a). An useful way to understand CDe is describing it as extending CI to also cover the aspect of running the software in production. Unlike *continuous deployment*, CDe seeks merely to confirm that the software is always in a state where it *could* be deployed at any time with the press of a button (Humble, 2010b).

An important aspect of using CDe is that problems with the deployment pipeline or with running the software project in production can be noticed and fixed before the code is deployed to production (Humble and Farley, 2010). As a software project is tested, deployed and used continuously, it gains feedback from each step that the developers can use to evaluate their changes, as well as notice regressions (Olsson et al., 2012).

Advocates for continuous practices assert that building, testing and deploying code automatically and often results in shorter feedback cycles, quicker fixing of bugs, and less manual work in testing and creating release (Humble and Farley, 2010). Of the continuous practices, continuous integration seems to be practiced in the majority of tech companies. Adoption of continuous delivery and deployment is still in the minority.

(Yang, 2019)

2.2.2 Tools for CI, CDe & CD

Organizations using continuous practice rely on tools for automating parts or most of the process. Tools for version control, automated builds and testing, as well as deployment are practically necessary for continuous practices to scale for larger teams and projects. Figure 2.3 visualizes how the tools relate to software development and release, as well as each other.

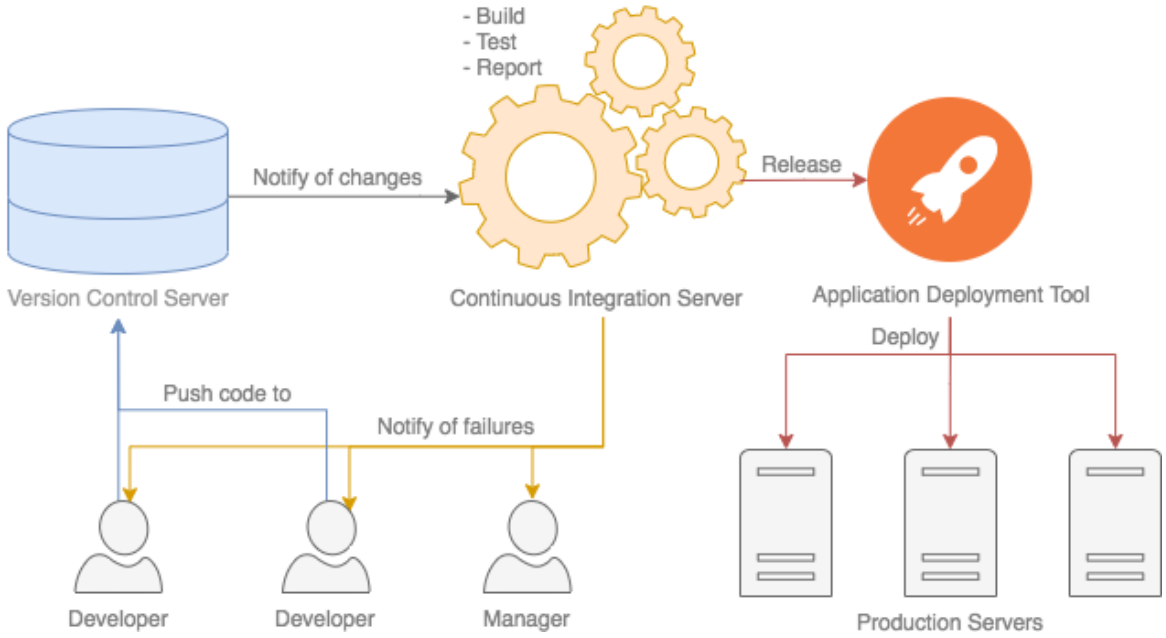

Figure 2.3: Version Control, CI and Deployment Automation tools and their role in project development and release.

Version Control tools, such as *git* (Chacon, 2019), *subversion* (Apache, 2019) and *Mercurial* (Mercurial, 2019) are important tools for practicing continuous integration. They automate and abstract away large portions of conflict resolution between different changes to the same parts of a code base. The most popular version control tool, git, uses branches to organize simultaneous development, with each branch representing a different set of changes applied to the code. Branches can be stored locally, or in a remote repository on a dedicated server. The main branch, to which most development is done, is usually called *master*.

Continuous integration tools are often woven into the version control tool. Continuous integration services such as *Travis* and *Jenkins* can launch a CI/CD pipeline for

each push to a version control remote. Cloud-based version control services, such as *GitHub* (GitHub, 2019) and *GitLab* (GitLab, 2019) support integrating build and test automation directly to the service providing version control. Using CI tools, CI/CD pipelines can either be run for all changes pushed to version control, or at scheduled times. Running pipelines often allows easily finding the set of changes that caused a build or a test to fail.

Application deployment tools such as *Ansible* (Hat, 2017) significantly reduce the costs involved with practicing continuous delivery and deployment. A large problem with manual deployment processes is the server drift that comes from manual changes to servers that are not consistently applied across the server instances. Using Ansible, projects are encouraged to keep infrastructure configuration as code, making server configuration and deployments predictable and reproducible. (Parnin et al., 2017)

Monitoring tools are an important part of practicing continuous deployment or delivery (Fitzgerald and Stol, 2014). They can be used to monitor the state of deployed applications, often providing information about things such as memory and processor usage. Often, any logs that applications write out are also monitored for errors. Based on the monitored state, alerts can be set up. With sufficient monitoring, problems in updated instances can be detected quickly.

2.3 Data integration

Data integration is the act of creating an unified view of data residing in different sources (Lenzerini, 2002). Software processes can be reliant on data residing in many different sources. Quite often, these sources store and output the data in different formats, and the data stored might be inconsistent between two sources. In this case, data integration is necessary to present the underlying data in an unified format, as well as to detect and remove inconsistencies in the data.

Systems for which the main responsibility is data integration are called **data integration systems** in the context of this study. A high-level example of a data integration system is visualized in Figure 2.4.

Data integration becomes necessary in situations, where a software service is reliant on data that is created elsewhere. Consider for example a cloud service that calculates how much a retail store needs to restock each of their products to keep up with demand.

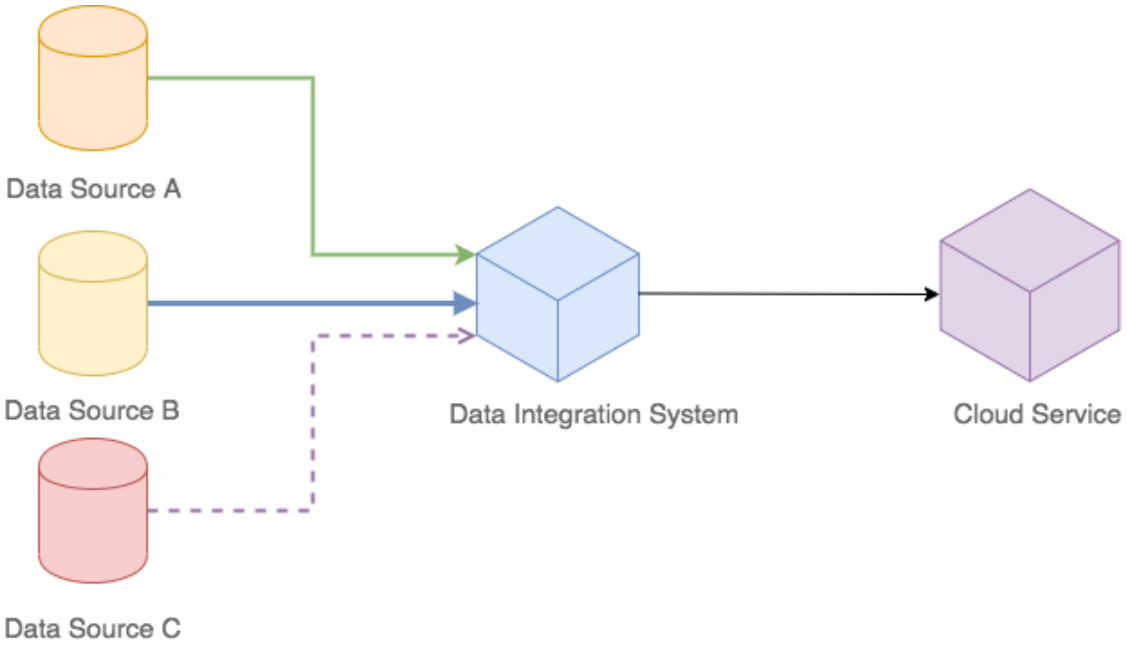

Figure 2.4: Data integration systems create an unified view of data originating from different data sources and potentially obtained through different data transfer methods.

To determine the demand, the service would require history data of which products were purchased at which times. This data needs to be read into the service.

In this example, the required data might be stored in various warehouse services. The data might also be in different formats depending on the product, with no format matching what the cloud service expects as input. In this case, data integration between the warehouse services and the cloud service is necessary. A system for doing this automatically would be called a data integration system.

3 Research approach

The thesis was conducted as a design science study (Hevner et al., 2004). Through working on the design and implementation of a software project at a Finnish software company, various problems and opportunities were identified with regards to the project's testing and deployment practices.

Using surveyed literature on the subject as well as existing domain and software engineering knowledge as a basis, we formulated acceptance criteria for a new end-to-end testing artifact. The design was implemented and evaluated in use at the case company. By evaluating the designed artifact, and contrasting it to the literature surveyed, we explored the requirements, benefits and challenges in adoption of continuous end-to-end testing.

Next, we will describe the case project on a high level. After that, the research questions of the study will be presented. Finally, the high-level acceptance criteria for the design are detailed.

3.1 Case project

The project that the case design was implemented for is a data integration system. It is responsible for receiving data from customer data stores and converting the data to a format accepted by the company's internal cloud service. The system is developed by a team of 5-10 developers. It serves as an inbound interface for retail customer systems, with the customers' sizes ranging from ones with revenues of billions of dollars to smaller companies.

Data received from the external services can vary in format, as well as in the method by which the data integration system receives the data. Via the data integration system, new interfaces through which data is received can be configured. For each interfaces the format and the type of the data it receives can be configured. The method through which data is received can also be customized for each interface.

The company's internal cloud service is reliant on a daily data flow from the customers' data stores. To be able to produce accurate results, the core service needs an unified

view of the state of the external services. Because of its importance, it is crucial that the data integration system is without bugs. Thorough testing is needed to ascertain that each individual release of the case project still fulfills the core requirements regarding its communication with and through other services.

3.2 Research questions

It is clear that the case project's communication procedures need to be thoroughly tested. Existing literature and software engineering practices would seem to indicate that end-to-end testing as one approach for gaining confidence in the communication. However, it seems that while end-to-end testing fulfills a proper niche in allowing testing of areas beyond the granularity level of lower-level tests, there are also many disadvantages associated with developing and maintaining end-to-end tests. As such, we set out to research the following research questions:

RQ1: What requirements are there for successfully adopting continuous end-to-end testing?

RQ2: What advantages and challenges does continuous end-to-end testing present?

By answering these questions, we hope to make it easier to evaluate whether continuous end-to-end testing should be adopted for any given software system.

3.3 Acceptance criteria

The needs and challenges identified in the case project were used to formulate acceptance criteria for an end-to-end testing artifact. Based on the literature and the practical needs, the following acceptance criteria were drafted.

Increase confidence in code. One of the primary purposes of software testing should be to increase confidence in the code. When portions of a project's features are untested, it becomes harder to make changes, as there is no indicator for when new code breaks existing features. As such, the artifact should cover areas that were previously untested, and increase confidence in the parts of the code base concerning communication.

Automatic and continuous tests. Continuous practices assert that testing should

be largely automatic. In addition, tests should be run often. With automatic and continuous testing, the test results are more reproducible, and provide quicker feedback to developers. In addition, there is less need to use time on manual testing. The artifact should thus be an automatic testing process.

Focused testing. The testing artifact should not try to test everything, rather the focus should be on the areas that end-to-end testing covers best, namely communication and collaboration between services. Areas that might make sense to include in end-to-end tests can opportunistically be added to the scope. However the first step to adding end-to-end tests should always be asking the question of whether the test should be placed on a lower level, where the test could be easier to maintain.

Integration to existing practices. The artifact should be as much of a natural fit to existing developer workflows as possible to ease its adoption. As such, the design should avoid introducing new tools or workflows unless necessary.

4 Case project

The case project, for which the end-to-end testing artifact was implemented for, is a data integration system named *Connect* of a Finnish software company. Connect is responsible for receiving data from customers' Enterprise Resource Planning systems and transforming it into a form readable by the company's *Software as a Service* supply chain planning solution, *Plan*.

In terms of supporting customer daily operations, Connect is an integral part of the process. Any bugs causing disruptions in the daily data flow to Plan can have far-reaching consequences for the solutions overall performance, and translate into real harm for the customer's business. Thus, assuring that the data integration between customer systems and Connect, as well as between Connect and Plan work correctly is important.

In this chapter, we give a brief description of how Connect is built and developed. From there, we discuss the high-level goals in building an end-to-end testing artifact, as well as give the practical requirements for it.

4.1 Architecture

With Connect, project teams can in collaboration with the customer define the interfaces through which Connect receives data. The interfaces describe how Connect receives data as well as what the data looks like. Connect can receive data from customers through a http web interface, as well as through transferring the files from a shared file server via *SFTP*.

A high level view of Connect's architecture is shown in Figure 4.1. Customers send data to Connect via the interfaces configured for it. Connect transforms the data, and reads it into Plan via its inbound interface, Runner. Connect instances and the Plan environment each Connect instance serves are located on the same server.

Depending on the transfer method, the data Connect receives can be represented in *CSV* or *JSON* formats. Connect additionally supports receiving data of multiple character encoding and compression formats. The different types of accepted data for

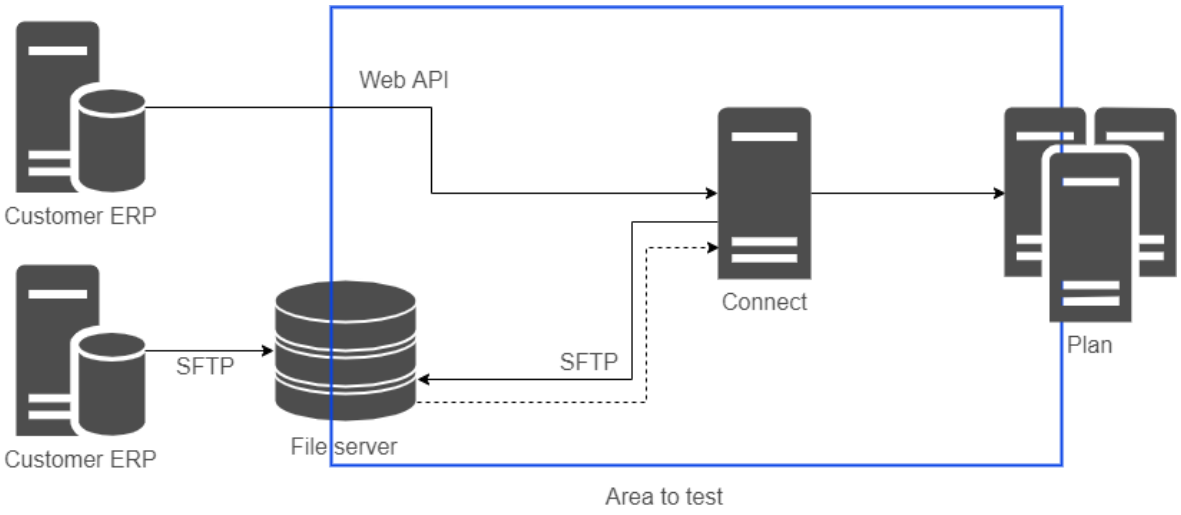

Figure 4.1: Simplified view of Connect architecture. Blue rectangle shows areas that would be covered by end-to-end tests.

Interface option	Web API	SFTP
Data format	JSON	CSV and JSON
Data content	Various	Various
Compression	gzip	zip, gzip, bzip2
Archival	N/A	zip, tar
Encoding	UTF, ISO, etc.	UTF, ISO, etc

Table 4.1: The various parameters affecting the kind of data Connect expects to receive.

both Web API and SFTP methods are listed in Table 4.1.

4.2 Existing continuous practices at case company

The Connect team follows continuous integration and delivery practices. Each commit to Connect's remote git repository launches a continuous integration pipeline for building and testing connect. New versions of Connect are also automatically deployed to an internal environment, as well as manually to production frequently. In this chapter, we detail how these steps are done.

4.2.1 Automatic builds and testing

Connect consists of a backend web server written in the *Kotlin* programming language, as well as a web browser frontend written written with a mix of statically typed *JavaScript* and *TypeScript*. The web server is tested with unit and integration tests written in *JUnit*. The frontend portion is tested using *jest*, as well as user-interface tests written in *Cypress*.

The backend and frontend tests are run automatically as part of a CI pipeline run with GitLab's dedicated CI/CD pipeline runner. Figure 4.2 visualizes the GitLab pipeline used to build, test and measure Connect's test coverage. On success, the pipeline deploys the build artifact to an internal test environment which is then restarted to run with the latest version. The artifact is also pushed to the case company's internal artifact repository

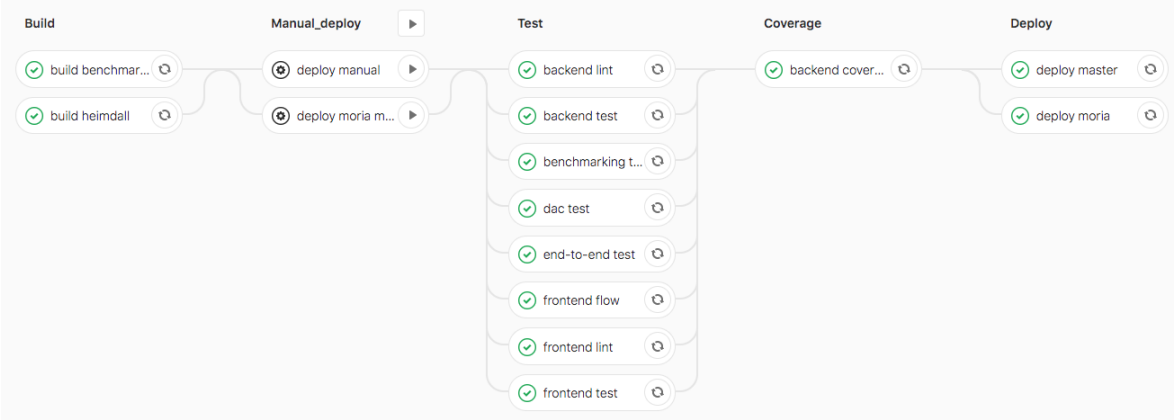

Figure 4.2: GitLab CI pipelines facilitate continuous practices with automatic tests and deployment to internal test environment.

4.2.2 Releases

Connect follows a bi-weekly release model. Every two weeks, a new release is made from the current state of Connect's master branch. As a part of each release, a change log is created from the changes since the latest release. The release process is manual. With each release, the version number is updated, and a new Connect release artifact uploaded to an artifact repository.

4.2.3 Deployment

Every two weeks, all Connect instances are updated to the latest release. An Ansible deployment script simultaneously updates all Connect instances' build artifacts to match the latest release artifact deployed to the company's internal artifact repository. Any instance-specific configuration files are also updated to their latest versions.

Before each release is deployed to existing Connect instances, a release schedule is announced to the project teams responsible for each customer instance. This allows the project teams to contact the customer and inform them of the downtime.

The release is deployed gradually. First, Connect is deployed to internal instances, as well as customer-specific test environment, used for testing customer-specific configurations before they're brought into production. After verifying that the test environments work as expected, the production environments are also updated to the latest release. In case of large regression resulting from the release, the instances can be restored to a previous version through use of daily backups.

4.3 Goals

Large parts of the data integration pipeline provided by Connect can be and are tested on lower granularity levels, using a mix of unit and integration tests. However, because of the scope of the tests, what can be tested is necessarily limited when compared to end-to-end testing. With end-to-end testing, it would be possible to cover the functionality that deals with communication between different services. Bugs caused by the difference between production and development environments would also be revealed by end-to-end tests run against a production-like environment.

It would be possible to test that the communication between different services works manually, perhaps using a mix of manual testing and scheduled data integration runs against a test instance. The manual testing could be improved further, by for example, more formally documenting the test cases that would need to be executed before creating each release.

However, testing these requirements manually quickly becomes time consuming as the amount of features to test as well as the number of different customer configurations rises. The testing being cumbersome has the effect of making it harder to have con-

fidence in each snapshot of the project working as intended. This causes a delay on when bug fixes and new features can be made available for customers, as the release process becomes more time-consuming.

Automatically testing the communication between Connect and Plan, and any auxiliary services, is thus needed. With automatic end-to-end testing, less manual work needs to be expended in testing each release, and the developers can gain quick feedback on when code they have written breaks the communication.

4.4 Requirements

In designing and implementing the end-to-end testing artifact, we identified a set of requirements it should fulfill.

REQ1: Test Connect & Plan interaction. The most important requirement concerns the interaction between Connect and Plan. Testing interaction between two services is not possible to test with lower-level tests, so testing it well via end-to-end testing is necessary. The end-to-end tests should verify that Connect writes its results to Plan in such a way that Plan reads them in correctly.

REQ2: Ensure Connect's inbound interfaces work in production. While Connect's interfaces can and are tested with lower-level tests, the lower-level test's results are not fully applicable to production environments. For example, Connect instances receive http requests through a http server that proxies the requests. Tests run locally however interface with Connect's http server directly. To cover this, and other cases where the production environment differs, the end-to-end tests should also verify that Connect's inbound interfaces work in production.

REQ3: Integrate to existing tools. The end-to-end tests should be a part of the daily workflow of developers. As such, the end-to-end testing artifact should be integrated to existing practices regarding testing and deployment. In Connect's case, this is the GitLab service, which is used for both source control as well as continuous integration.

5 Design

The designed test artifact is implemented as a scheduled step of Connect's GitLab CI pipeline. Using GitLab's scheduled pipelines, end-to-end tests are run against a Connect instance running on a dedicated test server that mimics a production environment.

Next, the testing strategy of the end-to-end tests will be discussed. Then, we will detail how the test environment for the end-to-end tests is set up. Finally, the way the artifact runs tests against the tested services will be detailed.

5.1 Testing strategy

The test artifact only tests so-called happy paths of the case project, meaning execution orders and inputs that are intended by the requirements. This is because testing invalid execution paths is easier on lower granularity levels, where the detail is sufficient to understand the edge and failure cases that might occur. On lower levels, the tests are also less flaky, and much easier to write. On a higher granularity level, the individual edge cases can often get lost in the noise generated by testing a larger subset of the system, and much more effort has to go into writing and maintaining the test cases.

As an example, testing that invalid input data received through Connect's interfaces produces validation errors is something that *can* be tested on a higher granularity-level. However, it is more naturally tested at a lower granularity, where the test is easier to write, and the testers are more aware of the specific cases that should be tested against the implementation. For higher granularity-level tests, the implementation becomes more of a black box, making it harder to understand what the important edge cases in the lower-level components are.

There could be cases where we'd want to test failure scenarios in communication. For example, in the future if Connect and Plan were to be run on different servers, we might need testing that verifies if failures in communication result in graceful behaviour. Even here though, these cases do not necessarily need end-to-end tests, but could rather be implemented with lower-level tests that stub Plan with a mock service.

5.2 Test environment

To produce reliable test results, it is important that the system-under-test is as close to production as possible. Connect and Plan should run the same versions as in a real environment, and they should be set up and configured in much the same way. Any auxiliary services, such as Apache Httpd, which serves as a proxy for http requests to Connect, should be included. The test instances for Connect, Plan, and all the auxiliary services they use are located on the same test server.

The test environment is set up by deploying Connect and Plan with their respective deployment tools. This is detailed in Figure 5.1. For Connect, the deployment is done using the Ansible deployment tool. The deploy sets up the configuration files for running and exposing Connect endpoints, as well as the Connect artifact itself. For Plan, the deployment is done using an internal company tool *Deploy*. Both services and their respective configurations are scheduled to be updated daily. The test artifact itself is deployed by GitLab's CI pipeline, which also ensures that the Connect artifact deployed is the latest version.

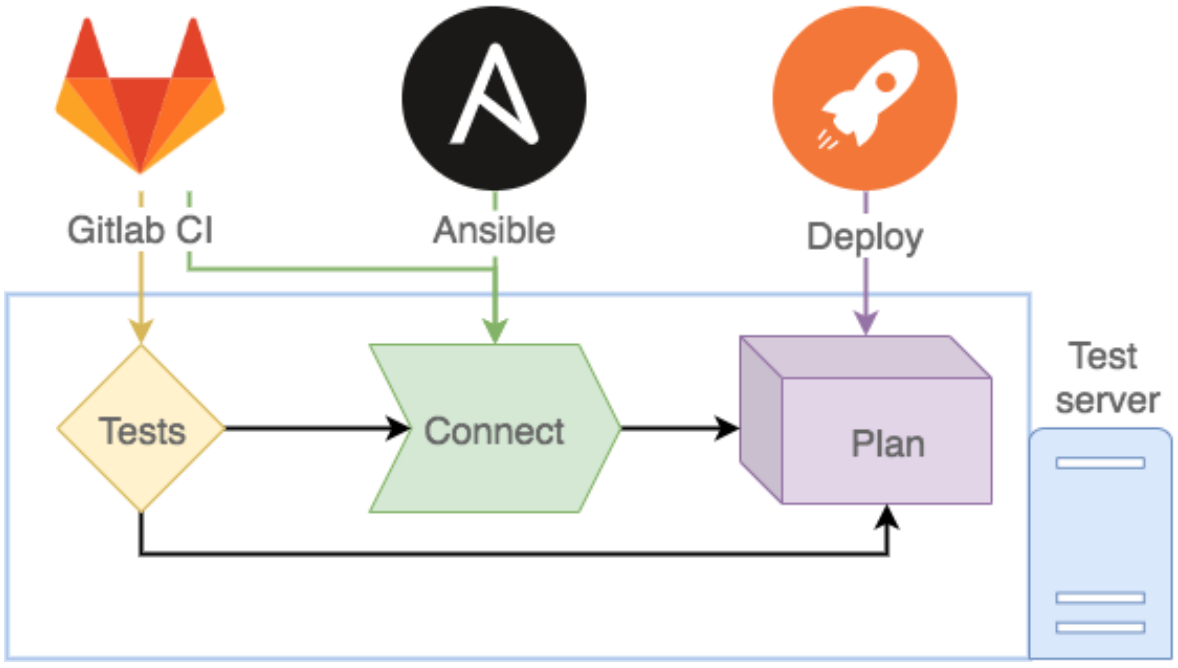

Figure 5.1: The end-to-end testing artifact is uploaded to and run on the test server. The tests are run against Connect and Plan instances, each deployed through their respective tool.

#	Step	Description
0	Setup environment	Start Connect instance with baseline configurations
1	Configure interfaces	Configure Connect for receiving the test data
2	Send data	Send data through configured interfaces
3	Monitor Connect	Monitor Connect for progress and errors
4	Verify results	Check and verify the results in Plan
5	Produce test summary	Create a summary of the test results
6	Reset instances	Connect & Plan instance reset to empty state

Table 5.1: Steps required for running end-to-end tests against Connect

5.3 Test runner

On each test run, an testing artifact is deployed from a scheduled GitLab CI pipeline to the test server. The artifact is then run remotely from the same pipeline, with the results logged to the GitLab's CI results. On the server, the artifact interfaces with Connect's and Plan's public interfaces, running data integration pipelines against Connect, and verifying that the results are read correctly to Plan.

The low-level details related to running each test case are abstracted away by a **test case runner** interface. Based on the details of each test case, the test case runner configures the Connect instance tested against to accept the types of data via the method specified by the test case.

The steps required for running a single test case are detailed in Table 5.1. Using the test case runner interface, it is easy to implement new test cases, as the technical details of each test are abstracted away. The testing artifact allows defining and running abstract **test cases** against a Connect instance. Each test case is a combination of the test parameters introduced in Table 4.1.

5.4 Running tests

Figure 5.2 describes the behaviour of tests for Connect's Web API interface. With the Web API tests, Connect is configured to accept Json data through it's http endpoints. Figure 5.3 in turn describes the behaviour for FTP tests. The main differences between the tests are in the way that data is delivered to Connect, with FTP data being read

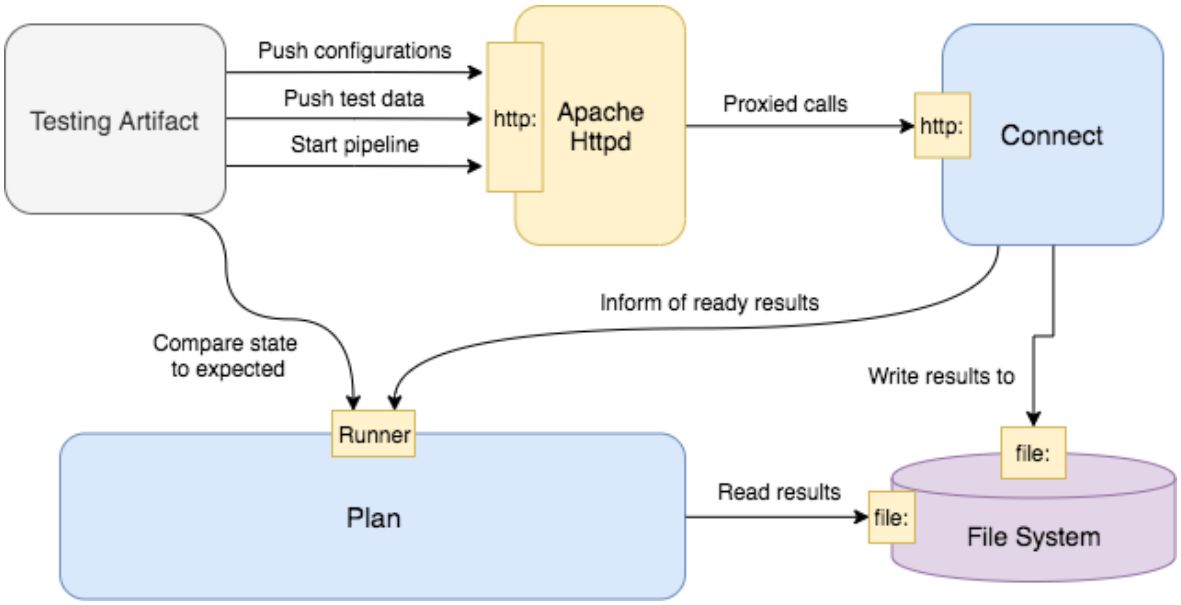

Figure 5.2: Web API Test architecture. Configurations and data are pushed to Connect through it's Json API, and the data is transformed and read to Plan.

by Connect from an FTP server when a Connect run is started.

Depending on the test parameters, the test artifact can apply different character encodings or compression algorithms to the test data sent to Connect. For example, in the case of the test case testing Json data encoded in UTF-32, and compressed with gzip, the data is first re-encoded, and then compressed before sending it to Connect.

The data received through Web API or SFTP is grouped, and a Connect run for transforming the grouped data is started. The data is transformed to a Plan readable format, and written to the file system. Connect will then notify Plan through its *Runner* interface that the data is readable, and start a Plan run for reading them in. On this being successful, the test artifact verifies that the data was read in correctly, and in case it has, marks the test case as a success. After a test case has been run, both Connect and Plan are reset to an empty state, so that each test case is run against a pristine test environment.

The interface that the end-to-end testing artifact uses for verifying that Plan received the correct results is Runner. Runner is a legacy interface for interfacing with Plan's internals directly. As such, it allows for great freedom in how it can be used, but is also susceptible to breaking in use when those internals are changed. As the number of potential use cases for Runner is huge, those use cases are also neither documented or tested.

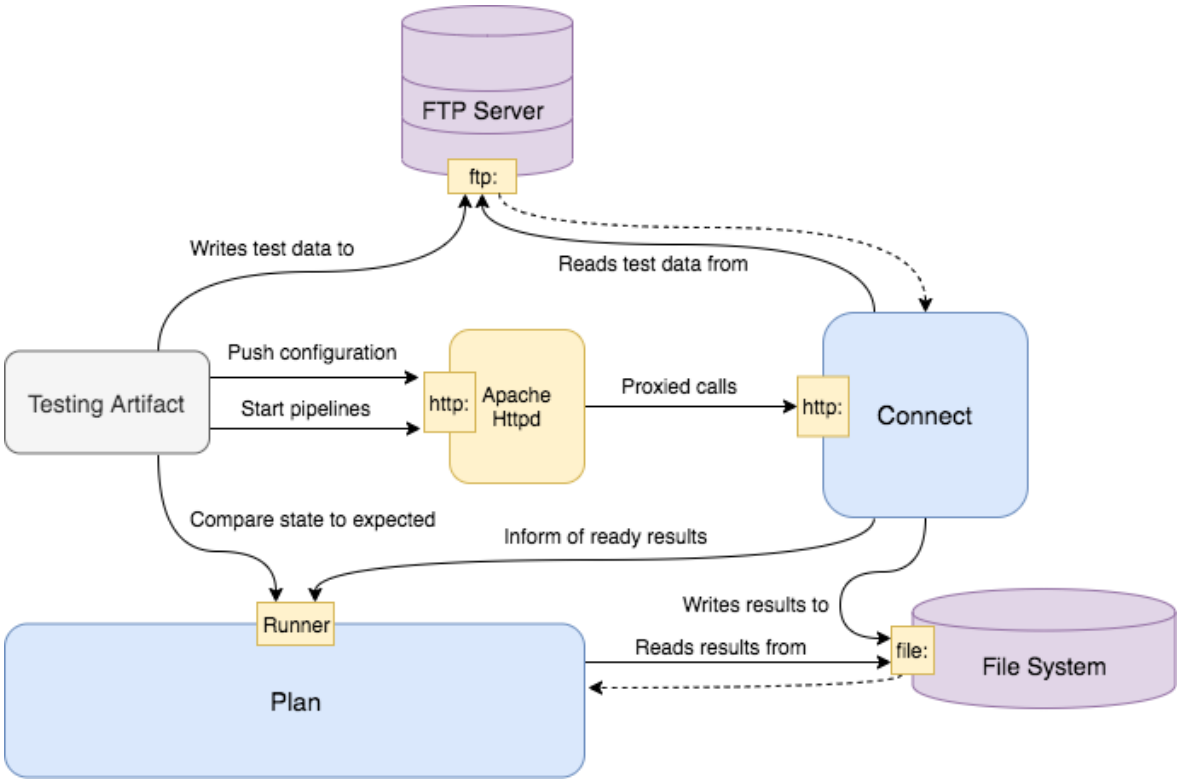

Figure 5.3: File Transfer Protocol Test architecture. Here, test data is written to an external file server, where Connect reads the data from.

During the test, the test artifact will validate that each step had expected results. In the case of failures at any of those steps, the test case is failed, and the failure case is logged. This makes it easier for developers to understand where failures happen, and makes the tests run quicker in the presence of failures. The test results, including which tests succeeded and failed, are logged to GitLab's CI user interface. On failure, the relevant Connect logs are linked in the logs to aid in debugging.

6 Results

The end-to-end testing artifact allows the communication between Connect and Plan to be tested continuously and automatically. However, challenges with the implementation were also identified, which can affect the artifact's usefulness in the future. Here, we list the results from implementing and using the artifact.

6.1 Findings

The end-to-end testing artifact set out to fulfill the requirements REQ1, REQ2 and REQ3. We found that each of the requirements was fulfilled. Additionally, other positive results were found, which we will present here.

6.1.1 Requirements

REQ1: Via the end-to-end testing artifact, it is possible to verify that happy execution paths against Connect result in expected data being sent to and read by Plan. As such, the interaction between Connect and Plan is now covered with automatic testing.

REQ2: The end-to-end tests are run against a Connect instance that is run in an environment identical to production. The tests verify that Connect is able to receive various forms of data through different transferal methods. The end-to-end testing artifact as continuous delivery. This allows for much quicker feedback than previous manual testing for when new changes break existing functionality in production environments.

REQ3: The end-to-end testing artifacts are a proper addition to the case project's existing continuous integration and delivery practices. By integrating the artifact with the tools already in use by the development team, the developers can see when changes they have made caused regressions.

6.1.2 Other findings

Reduced feedback cycle. We found that implementing end-to-end testing reduces the feedback cycle for developers. Previously developers would have to wait until acceptance testing formally started for results on if their changes broke communication in production. This could range from a few days to few weeks, depending on when the change was committed. Now, with the addition of daily automatic runs and continuous testing, this feedback cycle length has been reduced to a single day.

Easier to find breaking changes. As the tests are run continuously, it is now easier to find which change broke the tests. Running the tests each days limits the change set that the developers have to examine when tests are broken. This aids in detecting and fixing bugs in a timely fashion.

Less hours spent on testing. With the testing being automatic, less effort also needs to be spent on acceptance testing Connect. Previous testing procedures pushed data to Connect manually or via simple scripts. There, testers had to inspect manually that the data was processed by Connect successfully, and read in by Plan correctly. Having this process be automatic saves quite a bit of effort from acceptance testing.

Increased confidence. Previous knowledge of the status of communication between Connect and Plan was based on infrequently performed manual testing. Because of this, developers could be discouraged to make changes to the parts affecting communication. Developers could not make sure their changes did not break existing functionality. With end-to-end testing, confidence in the code, and consequently, the developer's willingness to make these changes has increased.

6.2 Challenges

On implementing the testing artifact, a few challenges were identified with regards to its use. These challenges can be grouped under the categories of *test environment setup* difficulties, *interface quality* and *test maintenance*.

6.2.1 Test environment setup

Orchestration of the test environment is one of the major pain points in developing end-to-end tests. The orchestration can extend to how the tests are run, as well as how

the test environment is set up.

Lack of reproducible test environment. Setting up the test environment for end-to-end testing between Connect and Plan is not a simple process. A large factor in this is that Connect and Plan have separate deployment processes and tools, which prevents developing an unified tool for deploying both. As such, the test environment is potentially hard to reproduce. If a test failure is not reproducible, due to its test environment having changed, or for any other reason, the failure will be that much harder to understand for the developers.

Lack of local test environment. Because of Plan's complex architecture and complex deployment process, a Plan instance running locally on a developer's workstation can not reproduce a production-like environment of Plan. As such, the testing artifact can *only* be used to test against a Plan running on a production-like server. New developments to the end-to-end test artifact itself can thus be burdensome to test. Running end-to-end tests requires deploying the test artifact to a test environment, which takes significantly more time and effort than if a local deployment for Plan was possible.

6.2.2 Interface quality

As the testing artifact relies on both Connect and Plan interfaces for its test execution and verification, any changes to those interfaces can mean that the end-to-end tests start failing. Here, we inspect the challenges that the interfaces of each service present to the success of the end-to-end testing artifact.

Connect interfaces. As the Connect interfaces used by the test artifact are not versioned, any changes on the Connect size can mean that test artifact will no longer be able to communicate with Connect. In this case, the test case being tested might still work, but the tests themselves are broken. In case changes to the interfaces are more common than failures in the cases being tested, the end-to-end tests could increase the burden on the developers, rather than decrease it.

Plan interfaces. Connect and the testing artifact communicate with Plan through interfaces that are not of the greatest quality. The interfaces have never had proper ownership inside the Plan development team. This has meant that the interfaces are largely undocumented and there is a lack of knowledge on how the interfaces are used.

As the use cases are largely not known by the Plan development teams, automated tests that verify these use cases work from version to version do not exist. Traditionally,

there has been a reliance on manual testing when new versions of Plan are released to customer environments. Before Connect, this was largely an issue for project delivery teams, who would have to rewrite their data integrations to work when breaking changes to Plan were made.

With the interface quality leaving something to be desired, there is a very real possibility that either Connect or the testing artifact's communication with Plan is broken by changes to Plan. This issue is compounded by the general lack of experience and knowledge regarding Plan in the Connect team. As Plan is a very complex software, with quite an unique architecture, the project can appear as a black box for Connect developers who have not previously worked with it.

It might be possible that the interfaces change infrequently enough that frequent test failures might not be that large of a concern. Improving the quality of the interfaces would, however, make future tests easier to develop and maintain.

6.2.3 Test maintenance

When implementing the design, challenges to the continued test maintenance of the end-to-end tests were also identified. These challenges have to do with test flakiness, as well as the high runtimes of the end-to-end tests.

Flakiness. For any test case that involves testing a large number of components working together, test flakiness becomes an issue. The more moving parts of Connect and Plan that are included in the test execution, the higher the chance that one of those parts starts behaving differently. This can mean test failures even when the system works correctly.

The tests are most likely to break when there are changes to the interfaces the artifact uses to communicate with Connect or Plan. However, even without changes to the interfaces, there might be changes to internal logic that could affect the test results. Alternatively, additional processes could be added or previous ones deprecated, which could change how the artifact should interact with Connect or Plan.

Test flakiness is something that will always exist when testing at a higher-granularity level. However, there are measures that can be taken to lessen the maintenance burden of flaky tests. Tests for the artifact itself would allow developers to more easily detect when changes to the artifact break tests. With integration testing, it might also be possible to detect when changes to Connect interfaces break end-to-end tests. In this

case, breaking changes could be fixed *before* the end-to-end tests are deployed and run.

Runtime. The runtime of end-to-end tests is rather large for a few reasons. The end-to-end tests require communication between Connect and Plan. They include potentially lengthy operations, and require resetting the services between tests. The tests are also run on a separate dedicated server, to which the test runner needs to be deployed. Additionally, since there is only one test environment, only one set of end-to-end tests can be run at a time.

A long test runtime can be harmful for the adoption of tests. If developers have to wait a long time for tests to be, it can lead to the developers' work being blocked as they wait for tests to finish, decreasing productivity. End-to-end tests are thus not feasible to run for each change, so a scheduled daily test run is used. With a smaller runtime, as well as ability to run multiple end-to-end tests in parallel, the end-to-end tests could become a part of the regular test suite, meaning faster feedback than scheduled tests.

Some optimizations could be achieved here by testing multiple parameters at once. The test cases are currently unique combinations of the parameters of test data type, format, encoding, compression and transferral method. This does not necessarily need to be the case, as it would be possible to test multiple values of some of the parameter types in a single test case. This particularly applies to the encoding and compression used.

6.3 Summary

The end-to-end testing artifact fulfills the requirements set up for it. The communication between Connect and Plan is tested. So are Connect's inbound interfaces. In addition, the tests are integrated to existing tools. Additionally, the results seem promising for making feedback cycles quicker, as well as decreasing manual work involved with testing.

However, there is a clear need to be mindful of the challenges presented, and take them into account when developing Connect and Plan further. Because of the many challenges in implementing end-to-end tests, the time put into the development was quite significant. There is a future risk, that if the challenges are not to be fixed, that automatic end-to-end testing can become too burdensome to maintain. This might discourage the developers from utilizing or developing the end-to-end tests further.

It can also be questioned whether end-to-end testing is the optimal approach to increasing confidence in the communication between Connect and Plan. End-to-end tests are the most vulnerable to changing requirements, and thus require much higher maintenance than other kinds of tests. If confidence in the communication between the system's services could be increased in other ways, end-to-end testing might not be needed.

7 Discussion

In this chapter, we will discuss the design and gathered results further, and how they answer the research questions we set out to study. The validity of the results are also detailed, and possible alternative approaches to the challenges discussed. Finally, recommendations for the next practical steps to take to improve the solution are presented.

7.1 Requirements for adoption of continuous end-to-end testing

In **RQ1**, we set out to research what requirements there are for adopting continuous end-to-end testing. Requirements concerning the system being tested, as well as the end-to-end test's implementation were identified and are listed here. With these results, teams will be be able to consider whether end-to-end testing is feasible for them at the moment, or if adoption would require investment into other parts of the system first.

7.1.1 System requirements

The results would seem to indicate that to successfully adopt continuous end-to-end testing, the system needs to satisfy certain requirements. The following requirements were identified:

- System consists of multiple independent services
- Mature public interfaces for interfacing with the services and verifying results
- Tools for deploying and keeping services up-to-date

Multiple independent services. Without multiple *ends* to test, testing at the granularity of end-to-end tests is not needed. In fact, testing at that granularity should rather be avoided, to keep tests from breaking often, and their runtimes low.

Mature interfaces. To be able to run test cases against the system's services, the services need *mature* public interfaces. Mature interfaces are interfaces that are either well-defined and versioned, or changed infrequently and battle-tested. Immature interfaces, which change frequently or produce unreliable results, have potential to cause large amounts of test failures, often not due to the case being tested failing, but due to the interfaces for configuration or verification working unexpectedly.

Deployment tools. For end-to-end testing, here needs to be a way to deploy and update each service included automatically. This is required for reliable test results, since manually deployed, often non-reproducible test environments cannot be trusted to produce the same results for the same tests later. End-to-end tests are already quite susceptible to flakiness by nature of testing many moving parts. Thus, it is important that the test environment does not further exasperate this problem.

7.1.2 Test requirements

To successfully adopt continuous end-to-end testing, the end-to-end tests themselves should fulfill certain requirements. The requirements identified were as follows:

- Limited number of test cases
- Abstraction of test implementation details
- Verbose failure logs
- Good accessibility of test results

Limited test cases. With end-to-end tests, it is important to understand the limitations and downsides inherent in testing at their level of granularity. It is in theory possible to test all functional requirements at the highest granularity. However, this would mean that each test case would necessarily be much more complex and require much more setup (Wacker, 2015). As such, the number of end-to-end test cases should be kept limited to only those that make sense to test on the granularity of multiple services.

Abstracting test implementation details. Often, testing at the level of independent services requires a lot of code for interfacing with each service. This can mean that, if the implementation details of each test are not abstracted away, it can become

harder to understand what each test is supposed to test, and how they fail. As such, it is useful to abstract the exact details of how the communication is done, separating them from the test case declaration. This can be done by specifying a domain-specific language for the tests, or by defining test cases as a combinations of test parameters. This makes defining and inspecting test cases easy, and exposes the relevant information about a test case to team members that might not be technically.

Verbose failure logs. With end-to-end tests, there are necessarily a large number of moving parts, each of which could cause the tests to fail. In the case of test failures, finding out which part caused a test to fail can be a larger undertaking than for lower-level tests, which better isolate the failure to the unit being tested (Fowler, 2013). Because of this, any end-to-end test should try to display where failures happen to the best of their ability, to aid in the debugging efforts of developers.

Test result accessibility. When implementing new processes to software development, their adoption is likely to be easier if the process and its results are accessible to the developers. This is the case for end-to-end testing as well. If end-to-end testing is made a part of the existing continuous practices and tools of a developer team, checking the test results can more quickly become a part of the developers' routine.

7.2 Advantages and challenges in continuous end-to-end testing

In **RQ2**, we researched the advantages and challenges for adoption of continuous end-to-end testing. The results indicate that adopting continuous end-to-end testing can be a valuable tool in testing systems where requirements are implemented through collaboration between multiple services. However, challenges inherent to end-to-end tests were also identified. The results should allow prospective adopters to more objectively evaluate how much they would benefit from adopting continuous end-to-end testing.

7.2.1 Advantages

Successful implementation of end-to-end tests confers tangible advantages to systems where they're adopted. The main advantages we identified are:

- Increased confidence in communication between services

- Reduced feedback cycle
- Testing collaboration between many services with complex communication patterns

Increased confidence. The results suggest that via end-to-end testing, confidence in the communication between independent services can be increased. With increased confidence, there is less need for lengthy manual testing processes and a reduced chance of bugs. The developers will also feel more confident in making changes that affect the communication, as they can gain accurate feedback on whether their changes cause tests to fail or not.

Reduced feedback cycle. With continuous end-to-end tests, the cycle between changes made and feedback received is shorter. This means that changes made can be validated faster, and improvements can be made based on feedback received while the changes are still fresh in a developer's mind. This is a considerable improvement over infrequent manual testing where a feedback cycle can take days or weeks. This matches with the goals of providing quick feedback in continuous integration (Fowler and Foemmel, 2006).

Testing collaboration between many services. It would seem that end-to-end tests are particularly useful in cases where communication happens between many services. If there are only a few services communicating with each other, the communication patterns might be relatively simple. For simple communication patterns, end-to-end testing might not be necessary, and an investment into interface quality might be more appropriate. As the communication and number of services becomes more complex however, end-to-end tests increase in value.

7.2.2 Challenges

While the advantages in adopting continuous end-to-end testing are clear, end-to-end tests also have significant challenges. Depending on the system being tested, these challenges might mean that different approaches should be considered. The challenges identified are:

- Large initial investment
- Low quality interfaces make developing and maintaining tests harder

- Large test runtime

Large initial investment. To transition from manual or partially automated testing to fully automated continuous testing, a large investment is needed (Olsson et al., 2012). As end-to-end tests often require complex test orchestration and environment setup, developing them can be very time-consuming. So the investment being large seems to apply doubly for end-to-end testing.

Low-quality interface maintenance burden. The results indicate that in cases where interfaces of the tested services are of low quality, developing and maintaining end-to-end tests is more burdensome. To measure interface quality, its documentation, test coverage, tendency to change and flakiness can be used. Undocumented interfaces are slow to develop against as much of the development is trial and error. Frequently changing or flaky interfaces can lead to test failures where the cause of the failure might be hard to understand.

Low-quality interfaces making end-to-end testing harder presents an interesting dichotomy. On one hand, when the interfaces are lower in quality, end-to-end tests might be needed to sufficiently verify that the interfaces continue to work. On the other hand, if the interfaces are of high quality, the need for end-to-end testing decreases as confidence in the communication is achieved through trust in the interfaces. However, since low quality interfaces will always exist, end-to-end testing does have its place in verifying their use.

Test runtime. The results indicate that the runtime of end-to-end tests is generally much larger than that of lower-level tests. As such, the number of end-to-end tests should be limited to avoid having to make the developers waste time waiting for test results. Large test runtime often means that it is unfeasible to make end-to-end tests a part of the regular test suite, meaning that results might only be reported once a day for example. This makes it harder to find out which change broke communication between different services.

7.3 Validity

There are some limitations in the study, which affect how generalizable the results are. The limitations concern the *observation period*, *limited data set* and *lack of literature*. Here we will discuss each of the limitations and how they might have affected the

results. Additionally, we shall discuss possible alternate approaches that could have been researched instead.

7.3.1 Limitations

Short observation period. As with all new processes, it is hard to evaluate the advantages and disadvantages in a short period of time. It could be that as the use of end-to-end tests matures as a process, the ways in which it affects the case project's practices become better understood. As the observation period for the finished design was quite short, some results that might become obvious later might have been missed.

With the observation period being short, some results were observed more in theory than in practice. For example, frequently changing interfaces are mentioned as a potential challenge for maintenance of end-to-end tests in the case project. However, it could be that the interfaces mentioned as problematic stabilize due to the effect of them now being tested, and the problem never manifests.

Limited data set. Since the practical research conducted was limited to a single company's project, the generalizability is necessary limited. As every software project and company us unique in their own way, the results might not be applicable to systems and teams with different concerns. It is also possible that the biases of the researcher have affected which results were considered important. Were a similar design made for a different company and project, it might result in different advantages and challenges being identified.

Lack of scientific literature. From surveying available scientific literature, it would seem end-to-end testing is a much larger topic in the practical software engineering field than in research. This might be caused by the particular testing methods employed by software projects being considered more of a practical and specific concern, with less generalizability to the field at large.

7.3.2 Alternative approaches

The results seem to indicate that there are cases where adopting end-to-end testing might not be appropriate. This can be because the system is not suitable because of its architecture. Alternatively the system might not be mature enough with regards to its interfaces or tooling. Another reason might be, however, that the advantages of end-

to-end testing are achieved in some other way. Here, we will discuss what alternative approaches to end-to-end testing could have been tried.

Contract testing. End-to-end testing can be supplemented or replaced with contract testing. Contract tests verify that a public interface fulfills a *contract*, specified via extensive documentation and test strategy, in how it behaves (Heckel and Lohmann, 2005). Services using the interface can then model their use of it against the contract. (Fowler, 2013)

Contract testing seems particularly strong in situations where collaboration happens between a limited number of services. In these cases, it is more likely that the communication flows between services are easy to map to different use cases. If sufficient documentation for those use cases exists and lower-level contract tests test that those use cases continue to work, developers are likely to be confident in the communication continuing to work.

Extensive monitoring. An another alternative for end-to-end testing is thorough monitoring of production instances (Clemson, 2018). Real load can be simulated against a production-like instance, with any errors during the process logged. With this, it is possible to quickly catch any issues in communication through monitoring the logs for errors. Provided sufficient logging and tested loads, this could provide confidence in communication between services.

Both contract testing and expanded monitoring could have been valid approaches to solving the challenges identified in the case project. However, they would also have required significant investment in the company either into higher quality interfaces or monitoring tools. While both of these are something that are being developed at the case company, the processes for their adoption were considered too slow or out of the hands of the developers for the case project. As such, these approaches were not studied.

7.4 Future work

The results identified many problems both in implementing and maintaining the end-to-end tests. Here, we propose some future practical steps and research that could be undertaken to alleviate some of the challenges identified.

Reproducible test environment. Lack of an easily reproducible test environment

is one of the main challenges identified in implementing the design. One way to improve in this area could be creating an easy way to run the whole system in a local containerized setup, via Docker (Docker, 2019) for example. With this, setting up a test environment for testing communication would be a simple process, and would not require orchestrating a flaky test environment on a test server.

Improved interface quality. An obvious recommendation for future practical work would be improving the interfaces of the case company's services. Public interfaces should be versioned, hide their implementation details and be well-defined enough that they serve as contracts between two services and their developers (De Souza et al., 2004). With high quality interfaces, there would be less need to rely on expensive end-to-end testing. One way to achieve quality interfaces could be a move towards microservice architectures.

Move towards microservice architecture. Architecting software as microservices has been claimed to result in clear ownership of interfaces, easier deployment, and cleaner separation of concerns (Nadareishvili et al., 2016). Adoption might thus result in good practices regarding interface design and implementation, as well as deployment. Going towards microservices could be one way to enforce a company culture where interfaces are designed and maintained in such a way that they're less prone to breaking, and individual services are easier to deploy for test and production.

However, it is at this point not clear what the specific advantages and disadvantages to adopting microservice architecture are. Neither is there a good understanding on how large of an investment it would require, and whether that investment would be worth it considering other priorities of the development teams. More research would be needed in this area to make an informed decision.

8 Conclusions

Based on a literature survey, as well as by designing and evaluating an end-to-end testing artifact in use at a case company, we identified requirements for projects to successfully adopt end-to-end testing. End-to-end testing's effectiveness in verifying communication between independent services was considered and contrasted with alternative approaches. Additionally, approaches that could help alleviate the challenges were presented.

The results indicate that to successfully adopt continuous end-to-end testing, systems and the test implementation have to satisfy certain requirements. Additionally, potential adopters should carefully consider whether end-to-end testing is the correct approach for their needs. End-to-end testing involves many challenges, and should not be considered a one-size-fits-all tool for testing. For systems with mature interfaces and tool-kits, it might be possible to verify communication between services using different approaches.

The research has furthered understanding of how and when end-to-end testing should be adopted for projects with multiple independent services. Challenges identified in the literature survey concerning end-to-end testing were confirmed in practice with the designed artifact. Additionally, the design shows how end-to-end testing can be introduced to existing continuous practices, with emphasis on quickly integrating it to the testing process of a software project.

For future practical work, investment into higher quality interfaces and environment setup tooling would alleviate many of the challenges identified in the research. It could be worth considering whether moving towards a microservice architecture at the case company could help in this. More research would however be needed to fully understand the advantages and disadvantages included in the approach.

References

Apache (2019). *Apache Subversion*. URL: https://subversion.apache.org/ (visited on 10/01/2019).

Chacon, S. (2019). *Git SCM*. URL: https://www.git-scm.com/ (visited on 10/01/2019).

Clemson, T. (2018). *Testing Strategies in a Microservice Architecture*. URL: https://martinfowler.com/articles/microservice-testing/ (visited on 09/17/2019).

Cypress (2019). *Cypress End-to-End testing framework - Documentation*. URL: https://docs.cypress.io/ (visited on 11/26/2019).

De Souza, C. R., Redmiles, D., Cheng, L.-T., Millen, D., and Patterson, J. (2004). "How a good software practice thwarts collaboration: the multiple roles of APIs in software development". In: *ACM SIGSOFT Software Engineering Notes* 29.6, pp. 221–230.

Docker (2019). *Definitive Guide to Enterprise Container Platforms*. Tech. rep. Docker, Inc. URL: https://www.docker.com/taxonomy/term/4955.

Fitzgerald, B. and Stol, K.-J. (2014). "Continuous Software Engineering and Beyond: Trends and Challenges". In: DOI: 10.1145/2593812.2593813.

Fowler, M. (2013). *BroadStackTest*. URL: https://martinfowler.com/articles/practical-test-pyramid.html (visited on 11/05/2019).

Fowler, M. and Foemmel, M. (2006). "Continuous integration". In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122, p. 14.

GitHub (2019). *GitHub*. URL: https://github.com/ (visited on 10/14/2019).

GitLab (2019). *Gitlab*. URL: https://gitlab.com/ (visited on 10/14/2019).

Hat, R. (2017). *Continuous Integration and Delivery with Ansible*. Tech. rep. Red Hat, Inc. URL: https://www.redhat.com/cms/managed-files/ContinuousDelivery-WhitePaper.pdf.

Heckel, R. and Lohmann, M. (2005). "Towards contract-based testing of web services". In: *Electronic Notes in Theoretical Computer Science* 116, pp. 145–156.

Hevner, R., March, S. T., Park, J., and Ram, S. (2004). "Design science in information systems research". In: *MIS quarterly* 28.1, pp. 75–105.

Humble, J. (2010a). *Continuous Delivery*. URL: https://continuousdelivery.com/ (visited on 10/01/2019).

Humble, J. (2010b). *Continuous Delivery vs Continuous Deployment*. URL: https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/ (visited on 09/17/2019).

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education.

Lenzerini, M. (2002). "Data integration: A theoretical perspective". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, pp. 233–246.

Mercurial (2019). *Mercurial SCM*. URL: https://www.mercurial-scm.org/ (visited on 10/01/2019).

Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc."

Olsson, H. H., Alahyari, H., and Bosch, J. (2012). "Climbing the" Stairway to Heaven"–A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software". In: *2012 38th euromicro conference on software engineering and advanced applications*. IEEE, pp. 392–399.

Parnin, C., Helms, E., Atlee, C., Boughton, H., Ghattas, M., Glover, A., Holman, J., Micco, J., Murphy, B., Savor, T., et al. (2017). "The top 10 adages in continuous deployment". In: *IEEE Software* 34.3, pp. 86–95.

Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., and Davia, B. (2002). "The impact of test suite granularity on the cost-effectiveness of regression testing". In: *Proceedings of the 24th International Conference on Software Engineering*. ACM, pp. 130–140.

Selenium (2019). *The Selenium Browser Automation Project*. URL: https://selenium.dev/documentation/en/ (visited on 11/26/2019).

Smeds, J., Nybom, K., and Porres, I. (2015). "DevOps: a definition and perceived adoption impediments". In: *International Conference on Agile Software Development*. Springer, pp. 166–177.

Wacker, M. (2015). *Just Say No to More End-to-End Tests*. URL: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (visited on 11/29/2019).

Yang, C. (2019). *DevTestOps Landscape Survey*. Mabl.